CHAPTER 4: FUNCTIONS AND PROGRAM STRUCTURE

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions can often hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make the use of functions easy and efficient. C programs generally consist of numerous small functions rather than a few big ones.

Most programmers are familiar with "library" functions for input and output (**getchar**, **putchar**) and numerical computations (**sin**, **cos**, **sqrt**). In this chapter we will show more about writing new functions.

4.1 Basics

To begin, let us design and write a program to print each line of its input that contains a particular "pattern" or string of characters. (This is a special case of the Unix utility program *grep*). For example, searching for the pattern "the" in the set of lines

```
Now is the time
for all good
men to come to the aid
of their party.
```

will produce the output

```
Now is the time
men to come to the aid
of their party.
```

The basic structure of the job falls neatly into three pieces:

```
while (there's a new line)
        if (the line matches the pattern)
                print it
```

If you put the code for all of this in the main routine, you'll create an unbelievable mess. (Try it.) A better way is to use the natural structure to

advantage, by making each part a separate function. Irrelevant details can be buried in the functions, and the chance of unwanted interactions minimized. Three small pieces are easier to deal with than one big one.

"while there's a new line" is **getline**, a function that we wrote in Chapter 1, and "print it" is **printf**, which someone has already provided for us. This means we need only write a routine which decides if the line contains an occurrence of the pattern.

We can solve that problem by stealing a design from PL/I: the function **index(s, t)** returns the position or index in the string **s** where the string **t** begins. We use 0 rather than 1 as the starting position in **s**, since arrays begin at position zero, and then **index** can return -1 if **s** doesn't contain a **t**. The **index** function centralizes some fairly messy logic in a single place, and provides a routine that may well prove useful in other contexts as well. (**index** wasn't written for this book — we wrote it for something else quite a while ago.) If we later need more sophisticated pattern matching we can **index** by a more general pattern matcher; the rest of the code remains the same.

Given this much design, filling in the details of the program is straightforward. Here it is in its entirety, including **getline**, so you can see how the pieces fit together.

```
main( )       /* find all lines matching a pattern */
{
        char line[MAXLINE];

        while (getline(line, MAXLINE) >= 0)
                if (index(line, "the") >= 0)
                        printf("%s\n", line);
}

getline(s, lim)       /* read line into s; return length */
char s[ ];
int lim;
{
        int i, c;

        for (i = 0; i < lim-1 && (c = getchar( )) != '\n' && c != EOF; i++)
                s[i] = c;
        s[i] = '\0';
        return(c == EOF ? -1 : i);
}
```

```
index(s, t)    /* return index of t in s, -1 if none */
char s[ ], t[ ];
{
        int i, j, k;

        for (i = 0; s[i] != '\0'; i++) {
                for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
                        ;
                if (t[k] == '\0')
                        return(i);
        }
        return(-1);
}
```

Burying the pattern to be searched for ("the") in the middle of main is not the most general of mechanisms, but we haven't yet discussed how to initialize character arrays; we'll return to this topic in due course.

Each function has the form

```
name(argument list, if any)
argument declarations, if any
{
        statements of function, if any
}
```

As noted, the various parts may be absent; a minimal function is

```
dummy( ) { }
```

which does nothing. (A do-nothing function is sometimes useful as a place holder during program development.)

A program with multiple functions is just a set of individual function declarations, as shown. Each function is self-contained, and the only communication between the pieces is (in this case) by arguments and values returned by the functions. The functions can occur in any order, and the source program can be split into multiple files, so long as no function is split. And of course the definition of EOF has to be accessible as getline is compiled.

The return statement is the mechanism for returning a value from the called function to its caller. Any expression can follow return, as in

```
return (expression)
```

It is common practice, though not required, to put parentheses around the *expression*, as we have done.

The calling function is free to ignore the returned value if it wishes. Furthermore, there need be no expresion after return; in that case, no value is returned to the caller. If there is no return statement in a function, control returns to the caller with no value when the function "falls off the end" by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another.

4.2 Functions Returning Non-Integers

So far, none of our programs has contained any declaration of the type of a function. This is because by default a function is implicitly declared by its appearance in an expression or statement, such as

```
while (getline(line, MAXLINE) >= 0)
```

In such cases, the context of a name followed by a left parenthesis is sufficient declaration. Furthermore, by default the function is assumed to return an int. Since char promotes to int in expressions, there is no need to declare functions that return char. These assumptions cover the majority of cases, including all of our examples so far.

But what happens if a function must return some other type? Many numerical functions like sqrt, sin, and cos return double; other specialized functions return other types.

To illustrate how this is done, let us write and use the function atof(s), which converts the string s to its floating point equivalent. atof is an extension of atoi, which we wrote versions of in Chapter 2 and 3; it handles an optional sign and decimal point, and the presence or absence of either integer part or fractional part. (This is *not* a high-quality input conversion routine; that takes more space than we care to use.)

To communicate the fact that atof returns float, the function itself must declare the type of value it returns, since it is not int. The type precedes the function name, like this:

```
float atof(s)  /* convert s to float */
char s[ ];
{
        float den = 1.0, val = 0.0;
        int i, sign = 1;

        for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
                ;       /* skip white space */
        if (s[i] == '+' || s[i] == '-')     /* sign */
                sign = (s[i++]=='+') ? 1 : -1;
        for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)        /* integer part */
                val = 10 * val + s[i] - '0';
        if (s[i] == '.')        /* decimal point */
                i++;
        for ( ; s[i] >= '0' && s[i] <= '9'; i++) /* fraction part */
                val += (s[i] - '0') / (den *= 10.0);
        return(sign * val);
}
```

Second, and just as important, the *calling* routine must state that this function returns a non-int value. The declaration is shown in the following rudimentary desk calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded by a sign, and adds them all

up, printing the sum after each input.

```
main( )        /* rudimentary desk calculator */
{
        float sum, atof( );
        char line[MAXLINE];

        sum = 0;
        while (getline(line, MAXLINE) >= 0) {
                sum += atof(line);
                printf("\t%f\n", sum);
        }
}
```

The declaration

```
        float sum, atof( );
```

says that sum is a float variable, and that atof is a function that returns a float value. Unless atof is properly declared in both places, C assumes that it returns an integer, and you'll get nonsense answers.

If atof and main are in the same source file, a type inconsistency will be detected by the compiler. But (as is more likely) if atof were compiled separately. the mismatch would not be detected, atof would return a float, which main would treat as an int, and garbage would result.

Given atof, we can write atoi (convert a string to int) in terms of it:

```
atoi(s) /* convert s to integer */
char s[ ];
{
        float atof( );

        return(atof(s));
}
```

Notice the structure of the declarations. Type conversion in return is a general rule — the value of the expression in

return(*expression*)

is converted to the type of the function before the return is taken. Therefore, the value of atof, a float, is converted automatically to int when it appears in a return, since the function atoi returns an int. (The conversion of float to int truncates any fractional part, as discussed in Chapter 2.)

As a further example, the following program computes the average and standard deviation of a set of one-per-line numbers. If a is the average of n numbers $x_1, \ldots, x_n$, the standard deviation is

$$\left(\frac{\Sigma(x_i)^2}{n-1} - a^2 \right)^{1/2}$$

The corresponding program is

```
main( )       /* avg and standard deviation */
{
        float atof( ), val;
        double dmax( ), sqrt( ), sum, sumsq, avg, std_dev;
        char line[MAXLINE];
        int n;

        sum = sumsq = n = avg = std_dev = 0;
        while (getline(line, MAXLINE) >= 0) {
                n++;
                val = atof(line);
                sum += val;
                sumsq += val * val;
        }

        if (n > 0)
                avg = sum / n;
        if (n > 1)
                std_dev = sqrt(dmax(sumsq/(n-1) - avg*avg, 0.0));
        printf("%d numbers: avg = %f, std dev = %f\n",
                n, avg, std_dev);
}
```

Here we have to declare sqrt to be a function that returns a double. The function dmax, which computes the maximum of two double's or float's, also has to be declared in main and written:

```
double dmax(x, y)
double x, y;
{
        return(x > y ? x : y);
}
```

The second argument passed to dmax is 0.0, not 0. Since dmax expects a double, passing it the int 0 could lead to disaster, or at least a wrong answer.

By the way, there is no entirely satisfactory way to write a function that accepts a variable number of arguments, because there is no portable way for the callee to determine how many arguments were actually used in a given call. Thus, you can't write a version of dmax that will take an arbitrary number of arguments, as will the MAX functions of Fortran and PL/I.

It is generally safe to deal with a variable number of arguments if the called function doesn't use an argument which was not actually supplied, and if the types are consistent. printf, the most common C function with a variable number of arguments, uses information from the first argument to determine how many other arguments are present, and what their types are. It fails badly if the types are not what the first argument says. Alternatively, it is possible to

mark the end of the argument list in some agreed-upon way, such as an "end of list" marker.

4.3 Arguments — Call by Value

In Chapter 1 we discussed the fact that function arguments are passed by value, that is, the called function receives a private, temporary copy of each argument, not its address. This means that the function cannot affect the original argument in the calling function (although if the argument was an array name, it can certainly affect array elements). Thus each argument is in effect a local variable initialized to the value the function was called with.

In Chapter 5 we will discuss the use of pointers to permit functions to affect non-arrays in calling functions.

4.4 External Variables

Speaking formally, a C program consists of a set of external objects, which are either functions or variables. Like functions, external variables are "global," that is, they are potentially accessible to any function. In this sense, external variables are analogous to Fortran COMMON or PL/I EXTERNAL.

Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may read or write data in an external variable merely by referring to it by name. For example, in the pattern-finding example shown earlier in this chapter, we could make the line buffer and the pattern external, and refer to them in main by including an appropriate extern declaration, like this:

```
char  line[MAXLINE];     /* line buffer */
char  pattern[ ]    = "the";     /* pattern to search for */

main( )
{
        extern char line[ ], pattern[ ];

        while (getline(line, MAXLINE) >= 0)
                if (index(line, pattern) >= 0)
                        printf("%s\n", line);
}
```

There are circumstances under which the extern declaration may be omitted, which we will discuss in a moment.

If a large number of variables must be shared among functions, external variables are more convenient than long argument lists, and will be rather more efficient. As pointed out in Chapter 1, however, this reasoning should be applied with some caution, for it can have a bad effect on program structure, and lead to program with many data connections between functions. Consider the pattern finding program again. Making line and pattern external is reasonable. It would be unreasonable, though, to write getline to store the input line

in a specific external variable; an argument is more flexible, since then **getline** can read into different arrays with different calls. And it would be a grave error to write **index** with the names of **line** and **pattern** built in instead of passed as arguments, since this would severely limit the utility of a general purpose routine.

A second reason for using external variables is that there are fewer restrictions on how they may be initialized. In particular, automatic arrays may not be initialized, but external ones may. In the pattern finding program, the array **pattern** can be initialized by the declaration

```
char  pattern[ ]  = "the";
```

if it is an external variable but not if automatic. We will treat initialization near the end of this chapter.

The third reason for using external variables is their lifetime. Automatic variables are internal to a function; they come into existence when the routine is entered, and disappear when it is left. External variables, on the other hand, are permanent. They do not come and go, so they retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, the shared data *must* be kept as external variables; there is no other way for it to be both permanent and accessible.

Let us examine this issue further in the context of another example. It is often the case that a program reading input cannot determine that it has read enough until it has read too much. One instance is collecting the characters that make up a name: until the first non-alphabetic character is seen, the name is not complete. But then the program has read a character that it is not really ready to deal with.

Dealing with this situation can tremendously complicate a program if we let it. Each time we need another character, we much check whether to read a new character or use the one we already have. Tangling this up with the logic of what to *do* with each character makes an unreadable mess.

The problem would be solved if it were possible to "un-read" the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read.

Fortunately, it's easy to simulate un-getting a character, by writing a pair of cooperating functions. **getc** delivers the next input character to be considered. **ungetc** puts a character back on the input, so that the next call to **getc** will return it again.

How they work together is simple. **ungetc** puts the pushed-back characters into a shared buffer — a character array. **getc** reads from the buffer if there is anything there; it calls **getchar** if the buffer is empty. There must also be an index variable which records the position of the current character in the buffer.

Since the buffer and the index are shared by **getc** and **ungetc** and must retain their values between calls, they must be external to both routines. As we saw in Chapter 1, a variable is external if it is defined outside of the body

of any function. Thus we can write getc, ungetc, and their shared variables as:

```
char  buf[BUFSIZE];     /* pushback buffer for getc and ungetc */
int   bufp  = -1;       /* current character position in buf */

getc( )     /* get a (possibly pushed back) character */
{
        extern char buf[ ];
        extern int bufp;

        return(bufp >= 0 ? buf[bufp--] : getchar( ));
}

ungetc(c)   /* push character back on input */
char c;
{
        extern char buf[ ];
        extern int bufp;

        if (bufp >= BUFSIZE)
                printf("Pushback overflow\n");
        else
                return(buf[++bufp] = c);
}
```

We have used an array for the pushback, rather than a single character, since the generality may come in handy later.

Exercise 4-1: Write a routine **ungets(s)** which will push back an entire string onto the input. Should **ungets** know about **buf** and **bufp**, or should it just use **ungetc**? □

Exercise 4-2: Suppose that there will never be more than one character of pushback. Modify **getc** and **ungetc** accordingly. □

4.5 Scope Rules

The use of external variables raises a number of questions about *scope*, that is, when variables are known to functions implicitly, and when declarations are needed.

The functions and external variables that make up a C program need not all be compiled at the same time: the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. The two questions of interest are

(1) How are declarations written so that variables are properly declared during compilation?

(2) How are declarations set up so that all the pieces are properly connected when the program is loaded?

C determines the scope of external variables by where they are defined in the source file. If an external variable is declared at some point in a file, then it may be referenced thereafter as if it had appeared in an **extern** declaration. For example, if **buf**, **bufp**, **getc**, and **ungetc** are declared in a single file, in the order shown above, that is,

```
char  buf[BUFSIZE];
int   bufp  = -1;

getc( ) ...

ungetc( ) ...
```

then there is no need for any **extern** declarations with **getc** and **ungetc**, and it is common practice to omit them.

On the other hand, if an external variable is to be used before it is defined, or if it is defined in a *different* file, then an **extern** declaration is mandatory.

There can only be one external *definition* of a variable among all the files that make up the source program; other files may contain **extern** declarations to access it. (There may also be an **extern** declaration in the file that defines it.) Any initialization of such a variable must go with the definition.

Thus, as an unlikely arrangement, but typical of larger programs, **buf** and **bufp** could be defined and initialized in one file, and the functions **getc** and **ungetc** defined in another. Then these definitions and declarations are necessary to tie them together:

In file 1:

```
char  buf[BUFSIZE];     /* pushback buffer for getc and ungetc */
int   bufp  = -1;       /* current character position in buf */
```

In file 2:

```
extern char buf[ ];
extern int bufp;

getc( )
{
        ...
}

ungetc(c)
char c;
{
        ...
}
```

Because the **extern** declarations lie outside both **getc** and **ungetc**, they apply to both; one declaration suffices for all of file 2.

4.6 Static Variables

Static variables are a third class of storage, in addition to the **extern** and **auto** (automatic) that we have already met.

static variables may be either internal or external. Internal **static** variables are local to a particular function just as **auto** variables are, but unlike **auto**'s, they remain in existence rather than coming and going each time the function is activated. This means that internal **static** variables provide private but permanent storage in a function. Character strings declared within a function, such as the arguments of **printf**, are internal static.

An external static variable is known within the *file* in which it is declared, but not any other file. External **static** provides a way to hide names like **buf** and **bufp** in the **getc-ungetc** combination, which must be external, yet which should not visible to users of **getc** and **ungetc**. If the two routines and the two variables are compiled in one file, as

```
static char   buf[BUFSIZE];     /* pushback buffer for getc and ungetc */
static int    bufp  = -1;       /* current character position in buf */

getc( )
{
        ...
}

ungetc(c)
char c;
{
        ...
}
```

then no other routine will be able to access **buf** and **bufp**; in fact, they will not conflict with the same names in other files.

Static storage, whether internal or external, is specified by prefixing the normal declaration with the word **static**:

```
static int ndigit[10];
```

The variable will be external or internal according to where the declaration occurs.

As a final note, functions may also be declared **static**; this makes them inaccessible outside of the file in which they are declared.

4.7 Register Variables

The fourth and final storage class is called **register**, which may be used to advise the compiler that the variables declared will be heavily used. The declaration is

```
register int x;
register char c;
```

and so on; the **int** part may be omitted.

In practice, only a few variables and only variables of certain types can be accomodated in registers on most machines. The specific restrictions vary from machine to machine; on the PDP-11, three register variables are allowed; they must be **int**, **char**, or pointer.

4.8 Block Structure

C is not a block structured language in the sense of PL/I or Algol and its derivatives. In particular, functions may not be defined within other functions; every function is external.

On the other hand, variables can be used in a block-structured way. Declarations of variables may occur after *any* left brace, not just the one that brackets the function. Variables declared in this way supersede any identically named variables in outer blocks, and remain in existence until the matching right brace.

The declaration of variables in a function is one instance of this. Given the declarations

```
int x;

f( )
{
        int x;
        ...
}
```

then within the function f, occurrences of x refer to the internal variable; outside of f, they refer to the external x. (Be sure that this is what you want; it can be hard to see this kind of error if it isn't.)

4.9 Initialization

Initialization has been mentioned in passing many times so far, but always peripherally to some other topic. This section summarizes some of the rules, now that we have discussed the various storage classes.

In the absence of explicit initialization, external and static variables are initialized to zero; automatic and register variables have undefined (i.e., garbage) values.

Ordinary variables (not arrays) may be initialized when they are declared, by following the name with an equals sign and a constant value:

```
int     x = 1;
char    c = 'X';
float   twopi = 2 * 3.141592654;
```

For external and static variables, the initialization is done once, conceptually at compile time. For automatic and register variables, it is done each time the function is entered.

For automatic and register variables, the initializer is not restricted to a constant: it may in fact be any valid expression involving previously defined values. For example, the initializations of the binary search program in Chapter 3 could be written as

```
binary(x, v, n)
int x, v[ ], n;
{
        int low = 1, high = n-1; mid;
        ...
}
```

instead of

```
binary(x, v, n)
int x, v[ ], n;
{
        int low, high, mid;

        low = 1;
        high = n - 1;
        ...
}
```

In effect, initializations of automatic variables are just shorthand for assignment statements.

"Aggregates", that is, arrays and structures, may not be initialized if they are automatic. External and static aggregates may be initialized, by following the declaration by a list of initializers enclosed in braces and separated by commas. For example, we can rewrite the character counting program of Chapter 1 from the original

```
main( )        /* count digits, white space, others */
{
        int c, i, nwhite, nother, ndigit[10];

        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
                ndigit[i] = 0;
        ...
}
```

to

```
int     nwhite          = 0;
int     nother= 0;
int     ndigit[10] ={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

main( )        /* count digits, white space, others */
{
        int c, i;
        ...
}
```

The initializations are actually unnecesary since all are zero, but it's good form to make them explicit anyway.

Character arrays are a special case of initialization. Since they occur so frequently, a string may be used instead of the braces and commas notation:

```
char  pattern[ ]     = "the";
```

It is also true that in all cases, the compiler will fill in the length of an array if there is a list of initializers. If there are fewer initalizers than the specified size, the others will be zero.

4.10 Recursion

C functions may be used recursively; that is, a function may call *itself* either directly or indirectly. One traditional example involves printing a number as a character string. The trouble is that the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. One is to store the digits in an array as they are generated, then print them in the reverse order, as we did in Chapter 3 with itoa.

```
printd(n)      /* print n in decimal */
int n;
{
        char s[10];
        int i;

        if (n < 0) {
                putchar('-');
                n = -n;
        }
        i = 0;
        do {
                s[i++] = n % 10 + '0';   /* convert last char to digit */
        } while ((n /= 10) > 0);   /* discard last digit */
        while (--i >= 0)
                putchar(s[i]);
}
```

The alternative is a recursive solution, in which each call of printd first calls itself to cope with any leading digits, then prints the trailing digit. The resulting code:

```
printd(n)      /* print n in decimal (recursive) */
int n;
{
        int i;

        if (n < 0) {
                putchar('-');
                n = -n;
        }
        if ((i = n/10) != 0)
                printd(i);
        putchar(n % 10 + '0');
}
```

Recursion generally provides no saving in storage, since somewhere a stack of the values being processed has to be maintained, nor will it be faster.

But recursive code is more compact, and often easier to write and understand. Recursion is especially convenient for recursively defined data structures like trees; we will see a nice example in Chapter 6.

Exercise 4-3: Adapt the ideas of **printd** to write a recursive version of **itoa**; that is, convert an integer into a string with a recursive routine. □

Exercise 4-4: Write recursive and non-recursive versions of the function **reverse(s)**, which reverses the string **s**. □

4.11 The C Preprocessor

C provides certain language extensionss by means of a preprocessor which is actually a rather simple macro processor. The **#define** capability which we have used is the most common of these extensions. The other aspect of the c preprocessor is the ability to include during compilation the contents of other files.

#include

To facilitate handling large collections of declarations (among other things) C provides a file inclusion feature. Any line that begins with

#include "*filename*"

is replaced by the contents of the file *filename.* Commonly a line or two of this form appears at the beginning of a file, to include common **#define** statements and **extern** declarations for global variables.

#include is definitely the preferred way to tie the declarations together for a large program. It pretty well guarantees that all the source files will be supplied with the same version of definitions and variable declarations, and thus eliminates a particularly nasty kind of bug.

Macros

A definition of the form

```
#define    YES  1
```

calls for a macro substitution of the simplest kind — replacing one fixed string by another. It is possible, however, to specify a macro with arguments, so the replacement text depends on the way the macro is called. As an example, we could define a macro called **max** like this:

```
#define    max(a, b)    ((a > b) ? a : b)
```

Then a line in a program like

```
x = max(p+q, r+s);
```

will be replaced by the text

```
x = ((p+q > r+s) ? p+q : r+s);
```

This provides a maximum "function" that expands into in-line code rather than a function call. So long as the arguments are treated consistently, this

macro will serve for any data type; there is no need for **max** and **dmax**, as there would be with functions.

Of course, if you examine the expansion of **max** above, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects. Some care has to be taken with parentheses to make sure the order of evaluation is preserved. And there are even some purely lexical problems: there can be no space between the macro name and the left parenthesis that introduces its argument list.

Nonetheless, macros are quite valuable. One practical example is the standard I/O library to be described in Chapter 7, in which commonly used functions like **getchar** and **putchar** are defined as macros (obviously **putchar** needs an argument), thus avoiding the overhead of a function call per character processed.

Exercise 4-5: Write macros for the functions **lower** and **upper** discussed in Chapter 2. □

Exercise 4-6: Write macros for **getc** and **ungetc** from this chapter. □